Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries

Nikhil Patnaik
University of Bristol
nikhil.patnaik@bristol.ac.uk

Joseph Hallett
University of Bristol
joseph.hallett@bristol.ac.uk

Awais Rashid
University of Bristol
awais.rashid@bristol.ac.uk

Abstract

Green and Smith propose ten principles to make cryptography libraries more usable [14], but to what extent do the libraries implement these principles? We undertook a thematic analysis of over 2400 questions and responses from developers seeking help with 7 cryptography libraries on Stack Overflow; analyzing them to identify 16 underlying usability issues and studying see how prevalent they were across the 3 cryptography libraries for which we had the most questions for on Stack Overflow. Mapping our usability issues to Green and Smith's usability principles we identify 4 *usability smells* where the principles are not being observed. We suggest what developers may struggle the most with in the cryptography libraries, and where significant usability gains may be had for developers working to make libraries more usable.

1 Introduction

Cryptographic APIs are hard to use. Other work has developed recommendations, guidelines and principles for how to make them more usable—but how can we tell when such usability recommendations, guidelines and principles are not being implemented? In this paper we focus on the ten principles proposed by Green and Smith [14] (reproduced in Figure 1). We investigate two key questions: (i) what are the issues that developers face when using seven cryptography libraries and (ii) what are the telltale signs that one of the ten usability principles is being violated?

Code smells are indicators that a piece of software code may be of lower quality than desired [12]. A code smell signifies that, while a piece of code may not be broken, it is violating a design principle and may be fragile and prone to failure. For example, Fowler defines the *Shotgun Surgery* smell as:

> "You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change." [12]

Code that smells of shotgun surgery may be correct and pass all the tests, but the smell suggests that there may be a deeper issue with the code's structure.

Following the idea of a code smell, a *usability smell* is an indicator that an interface may be difficult to use for its intended users. Past work has focused on usability smells in graphical user interfaces (GUIs)—indicators that end users may struggle to use an application [2, 16]. However, usability issues are not limited to GUIs. Developers struggle with programming interfaces in the same way that users struggle with user interfaces. For example, past work has suggested that improving the quality of documentation would lead to developers needing to ask fewer questions about how to use libraries [19]. If the developer is unfamiliar with the library they will rely on the documentation provided with the library and their own programming knowledge to implement the required cryptographic tasks. If we look at a developer question and answer site, such as Stack Overflow (a popular developer question and answer help website), we might expect to see fewer questions asking for help with the basic usage of a library if it has improved its API documentation. However, as our analysis shows, these smells are present across the cryptography libraries we examined and all can make usability improvements to help developers use them successfully.

In order to identify developers' struggles with cryptographic libraries, we analyze 2,491 Stack Overflow questions. We examine questions about seven cryptographic libraries (Table 1), selected for their popularity and to encompass a broad range of languages and use-cases. We conduct a thematic analysis [5, 8, 23] of the questions and answers looking for the underlying reason the question was asked—be that because of missing documentation, confusion around an API, lack of cryptographic knowledge, or developers preferring Stack Overflow to other resources. We identify 16 thematic issues across our corpus of questions and measure their prevalence across the different libraries. We relate these issues

Library	URL
OpenSSL	https://github.com/openssl/openssl
NaCl	http://nacl.cr.yp.to
libsodium	https://github.com/jedisct1/libsodium
Bouncy Castle	https://bouncycastle.org/java.html
SJCL	https://github.com/bitwiseshiftleft/sjcl
Crypto-JS	https://github.com/brix/crypto-js
PyCrypto	https://www.dlitz.net/software/pycrypto/

Table 1: Cryptography libraries examined in this paper.

back to Green and Smith's usability principles and identify *four usability smells* that indicate that *specific* principles are not being implemented fully. Finally we make suggestions, based on the prevalence of smells in each of the libraries, as to how library developers can better implement the principles to reduce the smells and make their API more usable.

The novel contributions of our investigation are as follows:

- An empirical validation of Green and Smith's principles showing when a principle is not being applied but also identifying issues that Green and Smith's principles currently do not capture.
- The thematic analysis of 2,491 Stack Overflow questions to assess the usability of cryptographic libraries.
- Identification of 16 thematic issues across 7 cryptographic libraries—capturing developers' struggles with regards to the usability of these libraries codified into four usability smells (Needs a super sleuth, Confusion reigns, Needs a post mortem, and Doesn't play well with others) which are signs that particular Green and Smith principles are not being fully implemented by a given library; before giving an overview of the prevalence of theses 16 issues, and 4 smells in 3 of the libraries (those that had over a hundred questions with a score ≥ 2).

2 Background and related work

The background and related work falls into two broad categories: research on usability issues of APIs in general; and work focusing on such issues in cryptography and security libraries. We discuss each of these bodies of work next.

2.1 Usability issues of APIs

Many studies have addressed the usability of APIs and why they can be difficult to learn and use. Zibran et al. [27] reviewed 1513 bug posts across five different repositories to identify the API usability issues that were reflected in the bug posts by the developers who used the APIs. They found 22 different API usability factors. We adopt a similar approach

Abstract Integrate cryptographic functionality into standard APIs so regular developers do not have to interact with cryptographic APIs in the first place.

Powerful Sufficiently powerful to satisfy both security and non-security requirements.

Comprehensible Easy to learn, even without cryptographic expertise.

Ergonomic Don't break the developer's paradigm.

Intuitive Easy to use, even without documentation.

Failing Hard to misuse. Incorrect use should lead to visible errors.

Safe Defaults should be safe and never ambiguous.

Testable Testing mode. If developers need to run tests they can reduce the security for convenience.

Readable Easy to read and maintain code that uses it/Updatability.

Explained Assist with/handle end-user interaction, and provide error messages where possible.

Figure 1: Green and Smith's 10 usable cryptography API principles, reproduced from [14]. We have given each principle a short name to allow easy reference throughout the paper.

and review the questions developers have about each one of our selected cryptographic libraries and see what prevalent usability issues arise. However, we investigate further to identify the usability smells from each library that contribute to the violation of the Green and Smith principles.

Other work has explored ways to measure the usability of existing APIs. Scheller and Kúhn proposed a framework for measuring an API's usability against a set of usability aspects [24]. Dekel and Herbsleb noted that many APIs place notes about when it is appropriate to use certain functions [9]. They developed a tool (eMoose) to integrate these notes into developer's editors (when using an annotated API) and found that developers who used their tool debugged programs quicker than those who only had access to the documentation. In follow up work [10] they noted that the key behind eMoose's success was not that eMoose made the notes immediately available, but rather that it helped provide a *scent* for programmers trying to debug their code—a hint that there was something that *could* go wrong and that prompted them to further read the documentation. Our work identifies *usability smells* that library developers and maintainers can use to understand how they may improve the usability of their libraries in line with the Green and Smith principles.

Helping developers avoid mistakes has been studied extensively with many papers suggesting ways APIs can be improved to help avoid mistakes and to speed up debugging when they inevitably occur. Bloch asserted general principles for API design that would produce a usable API [4]. These principles were summarized as 39 different maxims, though Bloch noted that good API design is a craft and couldn't be entirely captured by lists of rules. Others have proposed similar lists of metrics, often developed from studying or surveying developers. Ko and Yann studied the way developers use APIs [18]. They found that developers do not only need detailed worked examples but also good explanations of the concepts, parameters and ideas behind the API's design. Piccioni et al. [22] ran a study to assess the usability of an API by comparing the programmer's expectations to their performance. They found that issues with naming convention and types confused programmers, poor documentation made programming harder and that overly flexible APIs confused less experienced programmers. Clarke and Becker adapted the *cognitive dimensions* framework, used to describe the usability of user interfaces [15], to evaluate the usability of a class's API [6]. They suggested a list of ten dimensions that APIs should be judged on based on the original cognitive dimensions framework and two new ones that capture how much work any individual operation does. Our work complements such research by identifying the signs and smells – based on questions that developers ask when seeking help—that suggest developers are struggling to use an API.

As developers have changed so have their sources of documentation. Stack Overflow is increasingly used as a primary source of documentation. Parnin et al. surveyed questions about three popular (non-cryptographic) APIs on the site. They found that, on average, 80% of the API functions would be covered by at least one Stack Overflow question, but that only a relatively small pool of *experts* answered them [21]. Treude and Robillard looked at ways to extract insight from Stack Overflow questions and then how to integrate them with developer's toolchains [25]. In a small study of developers they established that developers found these extra insights helpful when programming. In a similar manner, our work studies developers' struggles through an analysis of the questions they ask on Stack Overflow. Our work adds to the literature by using Stack Overflow not to gain insight into developers, but rather into the usability issues of APIs the developers use.

2.2 Usability issues of cryptography and security libraries

Nadi et al. [20] examined over 1,000 Java cryptography-related questions and developed a set of 5 *obstacles* capturing the developer's problem based on the top 100 questions. Whilst Nadi et al. just looked at Java developers' struggles our work is broader examining developers' struggles with libraries for multiple languages and systems. They reported a low inter-rater reliability score (kappa = 0.41). We also relate these back to principles to identify underlying issues in the form of usability smells.

Egele et al. looked at the use of cryptographic APIs in Android applications [11]. They found mistakes in 88% of the apps that used cryptographic APIs. They developed a static analysis tool to identify mistakes automatically and proposed three usability guidelines to help developers avoid making mistakes in the future. Mindermann et al. studied the usability of cryptography APIs for the Rust programming language [19]. They noted that, whilst insecure defaults (and defaults in general) do not occur frequently in cryptographic libraries, very few projects warn about depreciated cryptography techniques or encourage developers to use more secure methods. They produced a list of 13 recommendations for cryptography APIs.

Various studies have assessed the usability of specific cryptographic libraries, e.g., [4, 14, 19] and developed a set of usability metrics. For example, one common guideline is:

> "Use a prominent location to link to the documentation, e.g., at the start page of the repository." [19]

Another guideline suggests providing examples so that developers can see how to use the library:

> "Example code should be exemplary. If an API is used widely, its examples will be the archetypes for thousands of programs." [4]

The issues we identify in this paper complement these guidelines by acting as usability smells—usability principles tell us how to make libraries more usable, the smells suggest where users are struggling due to such principles not being observed or implemented.

Studies have also shown that even if a cryptographic library is powerful developers may suggest to use an alternative cryptographic library which, although more usable, may be poorly implemented [1, 14]. For instance, Acar et al. [1] showed that the Python cryptographic library *Keyczar*, despite claiming to be designed for usability, was challenging to use because of poor documentation and lack of documented support for the key generation task. Surveying the developers after their study Acar et al. found that developers frequently found issues with missing documentation and examples in Python cryptographic libraries. We also find in our analysis of Stack Overflow questions that many developers struggle and ask questions due to issues with *missing documentation* and a need for *example code*, alongside several other issues.

If not all cryptography libraries are equally usable, then what issues do developers struggle with when using them? The analysis presented in this paper sheds light on such issues and identifies the usability smells that indicate that usability principles are not being fully observed in the design of a particular library.

3 Method

We investigate:

1. what issues with cryptographic libraries cause developers to seek help and ask questions on the Stack Overflow question and answer site;
2. how prevalent are the usability issues that we identify in seven cryptography libraries; and
3. what are the usability smells that are indicative of failures to implement Green and Smith's usability principles, and their prevalence in the 3 libraries for which we have sufficient data.

To answer these questions we selected seven cryptographic libraries to examine (Table 1), based on their prominence as well as to include a breadth of languages—C, Java, JavaScript and Python—and use-cases. Other libraries exist, but these seven cover a representative sample of what various cryptographic libraries currently look like including those with many users (OpenSSL) to those with only a few (SJCL). Additionally, two of the libraries (NaCl and libsodium) describe themselves as being *usable*, so we would hope to see a range of issues and differences between the usable cryptographic libraries and the not-so usable ones.

We scraped 2491 questions from Stack Overflow, using the library names as search terms and selecting only questions with a score greater than 1 to help avoid low-value and badly worded questions. Stack Overflow uses a reputation system to help combat spam—users with sufficient reputation[1] are allowed to vote for the usefulness of a question, which becomes the question's *score*.

We conducted a manual review of all 2491 questions: identifying the underlying issue, capturing common themes between questions, and verifying the validity of the answer. In order to do so, we studied the full description of the question on Stack Overflow to help us pinpoint the core theme of the issue. We also analyzed the explanations provided by other developers in the answer section to that question. The questions in our corpus cover a wide time period. Therefore, to address the issue of validity, once we studied the question and any related answers, we reviewed the prominent link of the cryptographic library to assess whether the question was still valid and unaddressed by the library. For example, if the developer said that they could not find documentation for a specific feature they wished to use, we checked if the documentation was still unavailable in the current version of resources for the cryptographic library. Questions that did not relate to an issue with the library itself, for example, due to users not understanding the behavior of their operating system's dynamic linker (we return to this issue in the Discussion; Section 7) or where the developer mistakenly attributed their question to a library, were not considered further. In total, we analyzed 2317 relevant questions.

We used thematic analysis, a qualitative research method used to extract themes from text [5, 8, 23], to identify recurring themes such as the need for documentation, build and compatibility issues within the Stack Overflow questions. We developed our themes by iteratively labelling questions, and then reviewing and discussing the labelling. We arrived at a set of 16 themes that captured the different issues developer's faced and ascribed a final, single theme to each of the questions we examined. Initially we used multiple themes, however we found only 4 cases where a question had multiple labels, so we simplified and ascribed the theme that best categorized the underlying reason the question was asked. We repeated the labelling with a regularly selected 10% subset of the questions analyzed by a second researcher and calculated Cohen's kappa—a commonly used measure of inter-rater agreement [7]. Cohen's kappa was 0.76 indicating that our coding was consistent between mappers.

3.1 Threats to validity

The questions in our corpus cover a period of several years. There is a danger that as time and the cryptographic libraries themselves change that the issues developers face could also change. To mitigate this we validated that usability issue identified in the library were still present in the current version. For example, if we attributed an issue to the documentation being missing, we validated that we still couldn't find the relevant documentation.

There is also the danger that an issue faced by a developer may be due to a particular problem faced solely by that developer and not a more general problem. To mitigate against this we selected questions which had a score greater than one—that is to say that more users of Stack Overflow believed the question to be worthwhile than not. Stack Overflow's reputation system is designed to help remove questions that have already been answered, and those that are of low-value (for example, questions where a developer has not asked a question, or questions where students are attempting to have their coursework answered for them). By selecting only questions with a positive score we help avoid some noise.

During our thematic analysis, each question was mapped to a single theme, with the dominant theme being picked in the case that a question could be attributed to multiple themes. For the most part, however, questions could be ascribed to a single theme and multiple themes were rare so a 1–1 mapping was used for consistency.

To identify the usability smells, we map the issues we identify to the usability principles that library developers should be implementing as identified by Green and Smith [14]. Various others have suggested different principles for developers (as we discuss in Section 2). We selected Green and Smith's principles because their principles have not currently been

[1] At the time of writing: 15 reputation to vote up, 125 to vote down.

validated, and were themselves a synthesis of other usability research [4] focused on usability and security issues. Other principles could be validated using the same methods and corpus as we have used however, and our dataset is available for comparative studies.

4 What usability issues do developers face?

Our thematic analysis reveals 16 usability issues with which developers struggle (Figure 2) categorized into 7 themes as shown in Figure 3. We discuss each of the issues and give some examples to demonstrate how they manifest in the questions posed by the developers.

4.1 Missing information

Missing Documentation. A developer states that they wish to use a function or form a feature that has components supported by the library but cannot find relevant information in the library documentation:

> "So I already know how to specify locations for trusted certificates using `SSL_CTX_load_verify_locations()`.[···] But nothing is mentioned about the trusted system certificates residing in the OPENSSLDIR."

Looking for Example Code. Not all library functionality needs an example, but it can be helpful to document common use-cases. The developer wishes to use a function supported by the library and requests examples of how the function is used. In the question the developer may address the quality of the example code or lack thereof:

> "I'm attempting to run:
>
> ```
> openssl pkcs12 -export -in "path.p12"
> -out "newfile.pem"
> ```
>
> but I get an error.
>
> ```
> unable to load private key
> ```
>
> How do I extract the certificate in PEM from PKCS#12 store using OpenSSL?"

This differs from *passing the buck* in that the developer has identified the functionality they want to use and made attempt at solving it. They have stated the problem they want to solve and have asked for a example in order to debug their own attempt.

Clarity of documentation. The developer found the documentation or output but found it vague or unclear in describing what exactly it does:

> "How can I interpret openssl speed output?
>
> I ran openssl speed on my Ubuntu computer. [···] what is 'Doing md4 for 3s' mean? does it mean do the whole test for 3 times/seconds? what does '1809773 md4's in 2.99s' mean? what does '8192 size blocks' mean? [···] And the above, last lines of openssl speed md4 output - what does they mean exactly?"

4.2 Not knowing what to do.

Passing the buck. The developer delegates their question to the Stack Overflow community, even though a quick search for the issue on the library website returns the answer needed:

> "I'm trying to convert the .cer file to .pem through openssl, the command is:
>
> ```
> openssl x509 -inform der -in
> certnew.cer -out ymcert.pem
> ```
>
> and that's the errors I'm getting:
>
> ```
> unable to load certificate
> ```
>
> What am I doing wrong?"

Rather than find the answer themselves the developer has *passed the buck* and used Stack Overflow to get the answer rather than search existing resources, as reflected in the response:

> "[···] like explained by ssl.com, a .cer file [···]"

Passing the buck differs from other issues, such as *what's gone wrong here*, in that the developer has made no attempt to solve the problem. They have encountered a problem and want someone else to give them the answer rather than work it out or find an existing solution by themselves.

Lack of knowledge. There were many instances where new users struggled with the functions provided by a library due to the lack of knowledge they had about the concepts of cryptography. For example:

> "[···] I'm using OpenSSL to avoid pay for it. I created my certificate this way: [···]
>
> But when I navigate to the website I get an "error" telling me that this is an "Untrusted certificate": The security certificate presented by this website was not issued by a trusted certificate authority."

This lack of knowledge is implicitly highlighted in the answer:

> "What you get from OpenSSL tool is a self signed certificate. Of course it is not trusted by any browser, as who can say you are worth the trust.
>
> Please buy a certificate if you want to set up a public web site [···]"

Missing Documentation. The cryptographic library does not have documentation available to address the issue.

Example Code. The developer asks for code examples to learn how to use a specific feature of the library or to learn how to implement some behavior. An example is either missing or lacking somehow.

Clarity of Documentation. The library has documentation for the developer's issue, but it is unclear or lacking additional information. The developer asks for clarification.

Passing the buck. The developer asks Stack Overflow, even though documentation regarding their issue has been given. Also questions where they ask a simple question which they answer themselves.

Lack of Knowledge. The developer does not have foundation level cryptography knowledge. The developer is new to cryptography as a subject and, in turn, the features of the cryptographic library.

Unsupported Feature. The crypto library does not support a security feature the developer wants to implement.

Borrowed Mental Models. The developer requests a mapping of a functionality between cryptographic libraries.

Abstraction Issue. Issues addressing the level of abstraction provided in the code of the cryptographic library. The developer wants a more detailed explanation than is provided by the documentation.

What's gone wrong here? The developer has code that looks like it should work, but fails—they are looking for an explanation why.

API Misuse. The developer has incorrectly used a specific feature from the cryptographic library.

Should I use this? The developer says what they wish to implement and asks which methods would be most apt to use.

How should I use this? The developer does not understand how to correctly use a feature or its various parameters.

Build Issues. Issues related to the setup of the cryptographic library and running provided tests.

Performance Issues. Issues regarding the performance of the cryptographic library.

Compatibility Issues. Issues related to integrating features from the cryptographic library with other libraries and tools.

Deprecated Feature. Issues addressing that a specific feature is not working, later to conclude that the feature is deprecated.

Figure 2: The 16 issues identified through a thematic Analysis of Stack Overflow Questions.

4.3 Not knowing if it can do

Unsupported feature. The developer wants to do something that the library does not support. This may suggest that the library is unclear about what it can and cannot do:

> "Has anybody Implemented ElGamal using OpenSSL or even inside?"

Borrowed mental models. The developer is trying to take a mental model about how one library works and apply it to different one:

> "How to recreate the following signing cmd-line OpenSSL call using M2Crypto in Python?:
>
> This works perfectly in command-line, I would like to do the same using M2Crypto in Python code.
>
> [⋯]"

The developer has tried to apply concepts from one library to another and has become confused when that doesn't work. This differs from *passing-the-buck* in that they are not unwilling to learn, they just don't know that the concepts differ. Passing-the-Buck is where a developer doesn't know how to use a library and tries to get someone else to tell them. They don't care about learning and just want to be told what to do.

4.4 Programming is hard

Abstraction issue. The developer needs help with an abstraction provided by the library. They've seen the documentation but they lack knowledge of the underlying abstraction to understand it. They need more help:

> I am trying to get my head around public key encryption using the openssl implementation of rsa in

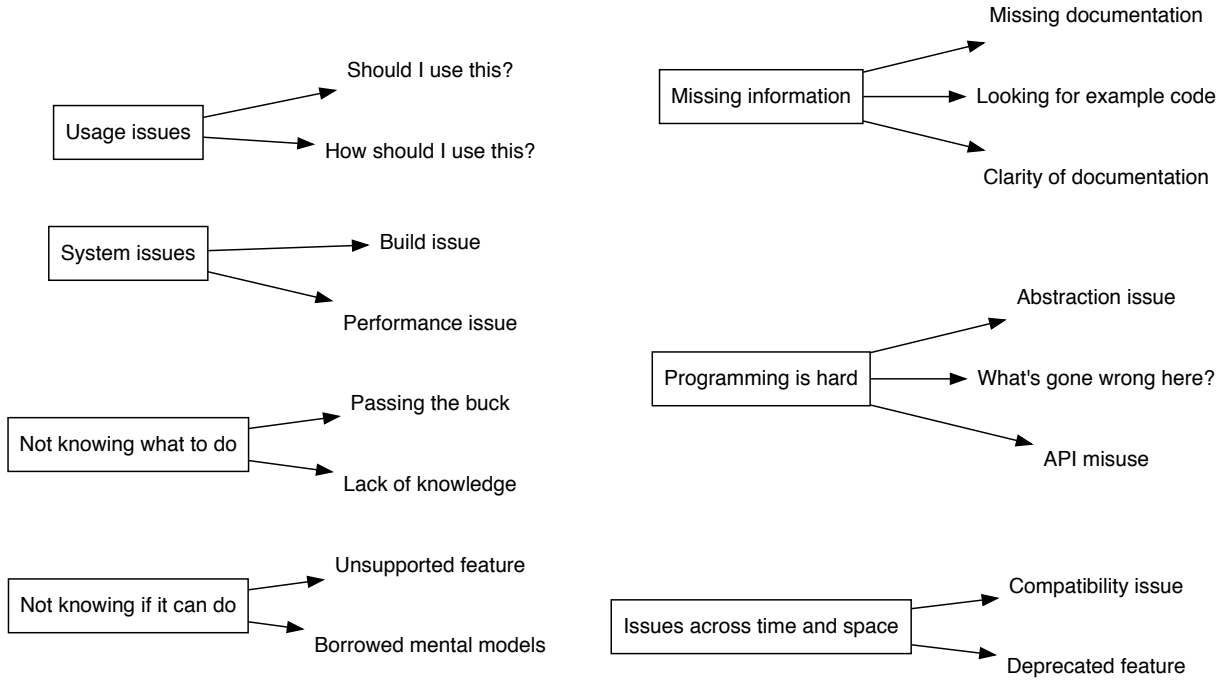

Figure 3: Categorization of the 16 issues identified through the thematic analysis.

> C++. Can you help? So far these are my thoughts (please do correct if necessary) [⋯] I see these two functions: [⋯] If Alice is to generate *rsa, how does this yield the rsa key pair? Is there something like rsa_public and rsa_private which are derived from rsa? Does *rsa contain both public and private key and the above function automatically strips out the necessary key depending on whether it requires the public or private part? [⋯]

What's gone wrong here? The developer has tried to use the library but has failed. They have given a specific example and asked Stack Overflow to suggest what has gone wrong:

> "Here is a certificate in x509 format that stores the public key and the modulo:
>
> ```
> const unsigned char
> *certificateDataBytes = /*data*/;
> ```
>
> Using OpenSSL and C, how can I convert it into an RSA object? I've tried several methods but I can't get it to work in `RSA_public_encrypt`"

API misuse. API Misuse represents questions where the developer incorrectly uses a function and they are corrected by another developer, usually supported with an explanation of the answer. For example:

> "[⋯] I'm trying to build a handshake protocol for my own project and am having issues with the server converting the clients RSA's public key to a Bignum. It works in my client code, but the server segfaults when attempting to convert the hex value of the clients public RSA to a bignum."

In the response to the question, the correct use of the function is explained:

> "RSA new() only creates the RSA struct, it does not create any of the bignum objects inside that struct, like the n and e fields. [⋯]"

4.5 Usage issues.

Should I use this? Developers have tasks and features in mind for which they want to know whether they should use a specific library function or not—or if there are two or more functions, which one should they use? In other cases, developers want to know whether the choices they make regarding the security of their application are appropriate:

> "I'm trying to build two functions using PyCrypto that accept two parameters: the message and the key, and then encrypt/decrypt the message.
>
> I found several links on the web to help me out, but each one of them has flaws:
>
> [⋯] Also, there are several modes, which one is recommended? I don't know what to use :/"

This differs from *missing documentation* where the developer is searching for specific API documentation, in that here they are unsure about which part of the API they want to use in the first place.

How should I use this? In contrast with *Should I use this*, in such cases the developer knows what they want to use, but is confused about some of the parameters involved:

> "How to compute RSA-SHA1(sha1WithRSAEncryption) value with OpenSSL?"

4.6 System issues

Build issues. To use an API developers must first build it (and run tests). This causes problems for the developer:

> "Error compiling OpenSSL with MinGW/MSYS"

Or

> "How to build OpenSSL to generate libcrypto.a with Android NDK and Windows"

Performance issues. The developer wants to use a library but finds that it isn't performant enough for their use-case. They seek help in optimizing their use of the library:

> "[···] profiling has revealed [···] 40% of my library runtime is devoted to creating and taking down HMAC_CTX's behind the scenes. [···] How do I get rid of the 40% overhead on each invocation in a (1) thread-safe / (2) resume-able state way? [···]"

4.7 Issues across space and time

Compatibility issues. The developer is struggling to integrate the library in question with another platform or library. For instance, out of the 2022 questions pertaining to OpenSSL, 244 were related to compatibility issues:

> "Encrypt in C# using OpenSSL compatible format, decrypt in Poco:
>
> I'm trying to encrypt (aes-128-cbc) in Win OS using a OpenSSL compatible format and decrypt on Linux OS using Poco::Crypto that is a wrapper of OpenSSL. "

Deprecated feature. The developer is trying to do something the library once supported, but doesn't know that the latest version has deprecated it:

> "After a few days of scouring the internet and openssl docs i've hit a wall [···]."

In the answers the developer realizes that they are using an outdated API.

> "Thanks to JWW and indiv i was able to solve my problem, it was an issue with me using older API's, and improper return checking. Solution: [···]:"

Issue	OpenSSL	Libsodium	NaCl	Bouncy Castle	SJCL	Crypto-JS	PyCrypto
Missing Documentation	256 (13%)	3 (9%)	5 (12%)	31 (17%)	4 (27%)	2 (6%)	7 (4%)
Example Code	128 (6%)		1 (2%)	10 (5%)	2 (13%)		4 (3%)
Clarity of Documentation	92 (5%)		3 (7%)	2 (1%)			6 (4%)
Passing the buck	136 (7%)	2 (6%)	4 (10%)	22 (12%)	4 (27%)	17 (49%)	10 (6%)
Lack of Knowledge	44 (2%)	6 (19%)	3 (7%)	19 (10%)		4 (11%)	17 (11%)
Unsupported Feature	24 (1%)		1 (2%)	5 (3%)			7 (4%)
Borrowed Mental Models	56 (3%)		2 (5%)	1 (1%)			
Abstraction Issue	40 (2%)		2 (5%)	2 (1%)	2 (13%)	2 (6%)	10 (6%)
What's gone wrong here?	259 (13%)	1 (3%)	2 (5%)	24 (13%)		3 (9?%)	16 (10%)
API Misuse	11 (1%)		1 (2%)	6 (3%)			7 (4%)
Should I use this?	84 (4%)		8 (19%)	19 (10%)		1 (3%)	8 (5%)
How should I use this?	80 (4%)			10 (5%)		2 (6%)	4 (3%)
Build Issue	362 (18%)	7 (22%)	3 (7%)	15 (8%)		3 (9%)	57 (36%)
Performance Issue	20 (1%)		1 (2%)				
Compatibility Issue	244 (12%)	7 (22%)	6 (14%)	8 (4%)	3 (20%)	1 (3%)	5 (3%)
Deprecated Feature	20 (1%)			9 (5%)			1 (1%)
Not Relevant	166 (8%)	6 (19%)		2 (1%)			

Table 2: Count of the number of Stack Overflow Questions attributed to each usability issue per library. Zero counts ommited.

5 How widespread are the issues across the seven libraries?

Table 2 shows the number of times each issue appeared during our thematic analysis for each cryptographic library; and suggests common issues across the libraries. *Missing Documentation* is a common issue: it suggests that developers face an issue in the first stages of using a cryptographic library as they are unable to locate documentation to support them. For instance, SJCL provides the code of each of its functions as its only developer support resource, and so can be made much stronger if they considered adding documentation to support the functions provided.

Passing the Buck and *Lack of Knowledge* highlight issues associated with developer behaviors instead of the cryptographic libraries themselves. Passing the Buck issues are common showing that developers have a tendency to pose questions on Stack Overflow, while the resources addressing the very questions are provided by the library and easy to locate. Many instances are recorded under the OpenSSL library, along with Bouncy Castle and Crypto-JS. Bouncy Castle and PyCrypto have a high percentage of questions associated with Lack of Knowledge. Developers address their lack of knowledge in their questions and request support in learning

cryptographic concepts in order to use functions from these libraries.

There are many occasions where the developer has a specific feature in mind for a project and wants to know how to securely implement this feature into the project. The reason the number of questions defined under *How should I use this?* is high for OpenSSL, for example, may be because the developer believes that other developers have already implemented the feature they had in mind. So the developer resorts to finding the specific implementation on developer community sites such as Stack Overflow instead of building their feature using the documentation provided by the cryptographic library. This could also explain why there are many questions where developers show an example of their broken code and request guidance with debugging. The answers usually come in the form of task-based examples of how to correctly implement, something to which the developers respond well.

Other than Missing Documentation, developers also highlight difficulties they have while setting up OpenSSL and running the provided tests. Reviewing the questions, we see that developers have projects in mind and intend to implement OpenSSL with other platforms they are using. This raises many questions associated with *Compatibility*. Developers find it very difficult to integrate OpenSSL with other platforms, a particularly pertinent issue as OpenSSL is widely used—and for large-scale projects. However, having compatibility issues makes OpenSSL less usable as developers cannot easily reconcile implementation of security requirements with other requirements for their projects.

6 Usability smells

Having identified the above issues, we map them on to Green and Smith's 10 principles (shown in Figure 1) in order to identify the usability smells that are indicative of one or more of the principles not being fully observed. The purpose of these smells is not to identify usability issues with a library early, but rather to guide work to improve a library's usability based on where developers appear to struggle between releases of a library as part of the software lifecycle. We note that Green and Smith's principles are written in a positive manner—for example:

> "Integrate cryptographic functionality into standard APIs so regular developers do not have to interact with cryptographic APIs in the first place."

In contrast, the issues we identify from the thematic study are written in a negative context—for example:

> "Missing Documentation: The cryptographic library does not have documentation available to address the issue."

The two viewpoints however are linked—if a library developer fails to fully implement a usability principle, then we might expect to see questions indicating that the library users are struggling with one of the usability issues we identify. Our mapping between usability principles and usability issues is presented in Table 3. For each issue we identified, we considered whether it would indicate failing to implement one of Green and Smith's principles. We did not map the *lack of knowledge* or *passing the buck* issues as these are attributable to specific developer behaviors and do not represent failures to implement usability within a library, and so do not map to Green and Smith's principles. For example the *borrowed mental model* issue is present when a developer expects one library to work similarly to another; this is mapped to the *ergonomic* principle as it indicates a failure to not break the developer's paradigm.

Based on this mapping, we identify four usability smells. In the same fashion as Fowler [12], we describe them as *whiffs*.

6.1 Needs a super sleuth

Issues at play: Missing documentation; Example code; Clarity of documentation.

You whiff this when documentation is missing, unclear or there is a lack of example code pertaining to how to use the library. The information to achieve the task you are intending to undertake is hard to find or understand in a way that can make the library work for your needs easily. You need to be a super-sleuth to find the documentation and decipher its meaning!

By not breaking the developer's paradigm (the ergonomic principle), developers can intuitively use the library with fewer references to the documentation or example code. By providing visible and early errors (the failing principle) developers can quickly understand when something is wrong and fix it themselves.

6.2 Confusion reigns

Issues at play: Should I use this; How should I use this; Abstraction issue; Borrowed mental models.

You can catch a whiff of this when developers are designing and prototyping their programs—they are trying to decide whether this is the right library to use and how to start using it. They are unclear as to how to use the library, perhaps having confused some concepts or borrowed a mental model they have for another library that isn't relevant here.

By making the library easy to use even without documentation (intuitive principle) a developer can quickly work out if they should use a library, how to use it and understand the abstraction it provides. If it is easy to learn (comprehensible principle) they can quickly evaluate it. If it uses standard APIs (abstraction principle) they can quickly figure out its use without worrying about details. By not breaking the developer's paradigm (ergonomic principle) they can reuse existing

Whiff	Issue	Abstract	Powerful	Comprehensible	Ergonomic	Intuitive	Failing	Safe	Testable	Readable	Explained
Need a super-sleuth	Missing Documentation				●						
	Example code				●		●				
	Clarity of documentation				●		●				
Confusion reigns	Should I use this?			●		●					
	How should I use this?	●	●			●					
	Abstraction issues	●				●					
	Borrowed mental models				●						
Needs a post-mortem	What's gone wrong here?				●	●				●	●
	Unsupported feature					●					
	API misuse						●	●			
	Deprecated feature						●				
Doesn't play well with others	Build issues								●		
	Compatibility issues		●								
	Performance issues										

Table 3: Mapping between developer issues and Green & Smith principles.

mental models about how similar libraries behave.

6.3 Needs a post-mortem

Issues at play: What's gone wrong here; Unsupported feature; API misuse; Deprecated feature.

If the *confusion reigns* whiff concerns the smells pre-coding, then this whiff occurs after they have written some code. The developer has used the library but something has gone wrong. Either they have used the library incorrectly or they are struggling to work out if it is an issue with the library itself. Perhaps an update to the library has broken their code, or led them to believe that it can do something it can't—either way the code needs a post-mortem.

By not breaking the developers model (ergonomic principle) developers can quickly guess whether their code is erroneous and work out what's gone wrong. If it is easy to use (intuitive principle) then this aids with debugging what's gone wrong as well as figuring out the capabilities and features of the library. Being hard to misuse (failing principle) avoids API misuse, and prevents API designers from deprecating APIs without a warning. If the defaults are safe and sensible (safe principle), then developers may avoid the complex API features and their potential misuses. Making the code easy to read (readable principle) means that if a developer needs to dive into the source to figure a bug out, they can do so with the minimum of fuss. Finally by helping developers with end-user interaction (explained principle), library designers can ensure developers do things in a standard way hence avoiding the need to ask if other people have done things the same way or differently.

6.4 Doesn't play well with others

Issues at play: Build issue; Compatibility issue; Performance issue.

If a library is going to be easy to use, developers have to be able to use it in the first place. This smell occurs when the library won't build, won't integrate with other libraries and build systems, and is a resource hog without providing a clear explanation why.

This smell doesn't appear to be particularly well covered by Green and Smith's issues. By adding a testing mode with only a subset of the features active (testable principle) developers can avoid having to build all the dependencies and get the library built for early testing and prototyping. By making the library powerful enough to satisfy security and non-security requirements (powerful principle) developers can more easily integrate it with other less flexible libraries.

We group performance issues under this smell, however we could not see any of Green and Smith's principles that exactly covered this usability aspect. In describing the *explained* principle, Green and Smith suggest:

> "Firstly, most developers using a security API do not have a firm grasp on the cryptographic or security background and thus would be hard pressed to explain to the end-user what went wrong" [14]

Perhaps by extending this principle to include not just *why things go wrong*, but also *why things take so long* this additional issue could be covered by Green and Smith's ten principles. Alternatively the *powerful* principle could be extended to cover not just the developer's primary security and non-security functionality requirements, but also cover the performance aspects.

Whiff	Issue	OpenSSL	Bouncy Castle	PyCrypto
Needs a super sleuth	*Whiffiness factor*	10% ●	11% ●	4% ●
	Missing documentation	13%	17%	4%
	Example code	6%	5%	3%
	Clarity of documentation	5%	1%	4%
Confusion reigns	*Whiffiness factor*	3% ●	6% ●	4% ●
	Should I use this?	4%	10%	5%
	How should I use this?	4%	5%	3%
	Abstraction Issue	2%	10%	6%
	Borrowed mental models	3%	1%	0%
Needs a post mortem	*Whiffiness factor*	10% ●	11% ●	8% ●
	What's gone wrong here?	12%	13%	10%
	Unsupported feature	1%	3%	4%
	API misuse	1%	3%	4%
	Deprecated feature	1%	5%	1%
Doesn't play well with others	*Whiffiness factor*	11% ●	5% ●	22% ●
	Build issue	18%	8%	36%
	Compatibility issue	12%	4%	3%
	Performance issue	1%	0%	0%

Table 4: What whiffs can you smell on each library? Percentages of the questions for each library that were mapped to each issue are shown, along side a *Whiffiness factor*, based on the weighted average, that indicates how strong the smell is:

●: particularly pungent (weighted average > 10%);

●: merest whiff (weighted average ≥ 2, ≤ 10%).

7 Discussion

With four whiffs established, Table 4 describes how smelly 3 of the crypto libraries appear to be—OpenSSL, Bouncy Castle and PyCrypto. For the remaining 4 libraries we lack a sufficient volume of questions to make any meaningful statement about the issues with which the library's users may struggle. However, for these 3 libraries we have 2,022, 185 and 160 questions respectively and so can consider where the *pain points* for developers using these libraries may lie. We include the libraries with fewer questions in our thematic analysis in order to reduce skew towards the issues prevalent in the more frequently queried libraries, however we lack the volume of questions required to suggest what the pain-points for developers are in the 4 remaining libraries. We do not claim that there is a fault in any of the libraries—rather we suggest what the most frequent issues that some developers struggle with when using them are—and where the biggest usability gains might be had. Future work should explore and find the underlying cause for the smell and establish *why* developers appear to be struggling.

For each library we add a *Whiffiness factor* (based on the weighted average of the percentage frequency of the issues associated with each whiff). All the libraries we looked at smell a little of *needing a super sleuth* with OpenSSL and Bouncy Castle users especially struggled with missing documentation. Despite this the overall whiffiness of this smell appeared to be low, as there were fewer questions over all 3 libraries associated with these issues—this suggests that documentation may be improving; and whilst documenting more of the library and giving more examples will help users, there may be bigger usability gains to be had elsewhere.

As for the *confusion reigns* whiff, again, the libraries all seem to show some signs of it—with the issue being particularly pronounced for Bouncy Castle, where we saw many developers asking whether it was appropriate to use this library, and having particular issues with the abstractions it provides. This is somewhat surprising as, at least for the Java version, Bouncy Castle integrates with the Java Cryptography Architecture which provides a standard API for libraries providing cryptography functionality. Bouncy Castle also provides its own API, and supports languages other than Java—perhaps offering too much choice confuses developer as to the parameters for a specific version. Focusing on the intuitive and comprehensible principles, i.e., by making the library easier to learn and understand without the need for expertise, should help reduce this smell.

The *doesn't play well with others* whiff was present for all three libraries. OpenSSL and PyCrypto in particular struggled with *build issues*, whereas Bouncy Castle (which is available as a precompiled JAR file) had fewer issues associated with building the library. Integrating software into systems is known to be difficult [13], but offering prebuilt images seems to go some way to mitigating this. Building the library is just the first step for OpenSSL however, as the library has to be linked into the final compiled program. When mapping the Stack Overflow questions, we saw several examples of developers asking about the dynamic linker. For instance:

> "I am using OpenSSL in my project.library is detecting but getting some errors like below:
>
> `Error:(23) undefined reference to 'RSA_generate_key' [···]`
>
> I included appropriate .so files in appropriate folder. I am not getting reason behind the undefined reference error.please help me to solve this issue."

These are not library usability issues as they represent a misunderstanding about the host-system and tools rather than the library itself. So we did not map them to an issue (the question in the specific example above was resolved by the developer updating their Makefile). The issue was common enough, however, that we believe that there may be a serious usability issue integrating libraries with systems, and a gap in the literature in looking into these issues. Further work is

needed to map out what these issues are, how common they are and what we can do to mitigate them.

The final whiff we identified, the *needs a post mortem* smell, was prevalent in the OpenSSL, Bouncy Castle and PyCrypto libraries. For these libraries the biggest contributing issue to this smell was that of developers trying to establish what had gone wrong with their programs. Making the code more readable and the libraries more intuitive to use even without documentation should help to make the debugging process easier and mitigate this smell. OpenSSL and Bouncy Castle are lower-level than other cryptography libraries providing greater access to their internals and crypto primitives. For these libraries we would expect an increase in the number of questions by developers trying to debug the code, simply because they wrap things up less into high-level APIs and offer more scope for developers to make a mistake. Perhaps then it is not unreasonable to expect lower-level libraries to display this issue more than the higher-level ones.

OpenSSL in particular, has been criticised in the past for being hard to use [1, 17, 26]. Kamp in particular argued for someone to:

> "Please Put OpenSSL Out of Its Misery.
>
> OpenSSL must die, for it will never get any better." [17]

Our analysis certainly suggests that OpenSSL is a bit stinky—in particular it seems that developers struggle a lot debugging it and in finding the documentation. Ignoring those issues, however, it is similar to the other crypto libraries and sometimes a little bit better (it seems better at abstraction, describing its parameters, for example)—at the very least both Bouncy Castle and PyCrypto appear harder to debug. OpenSSL gets a lot of stick for being unusable but perhaps it doesn't deserve it all—it has a general pong of poor usability but there are other libraries with sharper, more specific, stenches out there too.

8 Conclusion

How can we tell what a developer is struggling with when using a crypto library? Through our analysis of a substantial corpus (2491) of questions from Stack Overflow, we found 16 issues and four whiffs that suggest when developers are struggling. By linking these smells to the usability principles by Green and Smith [14], we can suggest how to improve crypto libraries and make them more usable for developers. Our study offers evidence to validate parts of Green and Smith's heuristics, but also highlights issues that were missed. Their usability principles suggest ways to mitigate most of the issues we identify; however issues associated with the *doesn't play well with others* smell (in particular *build* and *performance* issues) suggest the need for an additional principle to help cover these issues.

Our whiffs capture the general problems developers have when using crypto libraries. Not all libraries smell the same, and improvements to *usable* crypto libraries appear to be paying off with fewer usability smells. By smelling carefully we can find the pain point for developers and help improve usability. Libraries will perhaps always be a bit smelly given the challenges of catering for the requirements of a wide and diverse set of developers and applications; but by integrating usability principles we can at least make them less so.

9 Acknowledgements

This work is supported by funding from the National Cyber Security Centre and in part by the Engineering and Physical Sciences Research Council grant EP/P011799/2: Why Johnny doesn't write secure software.

References

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the usability of cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy*, pages 154–171, May 2017.

[2] D. Almeida, J. C. Campos, J. Saraiva, and J. C. Silva. Towards a catalog of usability smells. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 175–181. ACM, 2015.

[3] D. G. Altman. *Practical statistics for medical research*. CRC press, 1990.

[4] J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507. ACM, 2006.

[5] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[6] S. Clarke and C. Becker. Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*, 2003.

[7] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[8] J. W. Creswell and J. D. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 1994.

[9] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Proceedings*

of the 31st International Conference on Software Engineering, pages 320–330. IEEE Computer Society, 2009.

[10] U. Dekel and J. D. Herbsleb. Reading the documentation of invoked API functions in program comprehension. In *2009 IEEE 17th International Conference on Program Comprehension (ICPC 2009)*, pages 168–177. IEEE, 2009.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[12] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[13] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *1995 17th International Conference on Software Engineering*, pages 179–179. IEEE, 1995.

[14] M. Green and M. Smith. Developers are not the enemy!: The need for usable security APIs. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[15] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of visual languages and computing*, 7(2):131–174, 1996.

[16] P. Harms and J. Grabowski. Usage-based automatic detection of usability smells. In *International Conference on Human-Centred Software Engineering*, pages 217–234. Springer, 2014.

[17] P. H. Kamp. Please put OpenSSL out of its misery. *ACM Queue*, 12(3):20–23, 2014.

[18] A. J. Ko and Y. Riche. The role of conceptual knowledge in API usability. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 173–176. IEEE, 2011.

[19] K. Mindermann, P. Keck, and S. Wagner. How usable are Rust cryptography APIs? *arXiv preprint arXiv:1806.04929*, 2018.

[20] S. Nadi, S. Kriüger, M. Mezini, and E. Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[21] C. Parnin, C. Treude, L. Grammel, and M. A. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep*, 2012.

[22] M. Piccioni, C. A. Furia, and B. Meyer. An empirical study of API usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on*, pages 5–14. IEEE, 2013.

[23] J. Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

[24] T. Scheller and E. Kühn. Automated measurement of api usability: The api concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[25] C. Treude and M. P. Robillard. Augmenting API documentation with insights from stack overflow. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 392–403. IEEE, 2016.

[26] M. Ukrop and V. Matyas. Why Johnny the developer can't work with public key certificates. In *Cryptographers' Track at the RSA Conference*, pages 45–64. Springer, 2018.

[27] M. F. Zibran, F. Z. Eishita, and C. K. Roy. Useful, but usable? factors affecting the usability of APIs. In *2011 18th Working Conference on Reverse Engineering*, pages 151–155. IEEE, 2011.